Codex Practical Blue Book: From Beginner to Architect

Author: Hunk Wu (X: @ai_pmer)

🌐 中文 PDF 版 | 🌐 中文在线版

🧭 Table of Contents

Part 1: Product Survival in the AI-Native Era

- Ch.01 Saying Goodbye to Handwritten Code: Product Mindset in the Era of Vibe Coding
- Ch.02 Cross-Device Control: Building Your Codex Multi-Surface Productivity Matrix
- Ch.03 Breaking the Cloud Island: Sandbox Debugging and Deep Local Environment Tunneling

Part 2: Architecture & Constraints

- Ch.04 Goal-Driven Engineering: Taming Reasoning Agents with Boundaries and Assertions
- Ch.05 Defining the CAP Protocol: Building Your Project's AGENTS.md Rule Compliance Layer
- Ch.06 Correcting Course: Supervising the CoT Reasoning Chain Like a Tech Lead

Part 3: Advanced Multi-Surface Telemetry

- Ch.07 Closing the Visual Loop: Automated Auditing and Design Verification with Desktop Computer Use
- Ch.08 Mobile Sentinel Workflows: 24/7 Remote Development and Orchestration
- Ch.09 Codebase Revitalization: Reverse Engineering and Progressive Decoupling of Legacy Systems

Part 4: One-Person SaaS Commercialization

- Ch.10 Monetization in Practice: Shipping a Commercial SaaS MVP in 2 Hours
- Ch.11 Mobile Extension: Expo Cross-Platform App Development and Cloud Packaging
- Ch.12 The Final Frontier: Building an Automated Growth Flywheel for a One-Person SaaS

Part 5: Frontier Watch & Version Migration

- Ch.13 Frontier Watch: The 2026 Codex Ecosystem Overhaul

🔌 Related Projects

- [Feishu Assistant (Codex Feishu Sentinel)]: The official companion repository for Ch.08. An intelligent duty assistant in Feishu for Codex developers, featuring automated daily git digest pushes, CI mobile alarms, and one-tap remote approvals.

Ch.01 Saying Goodbye to Handwritten Code: Product Mindset in the Era of Vibe Coding

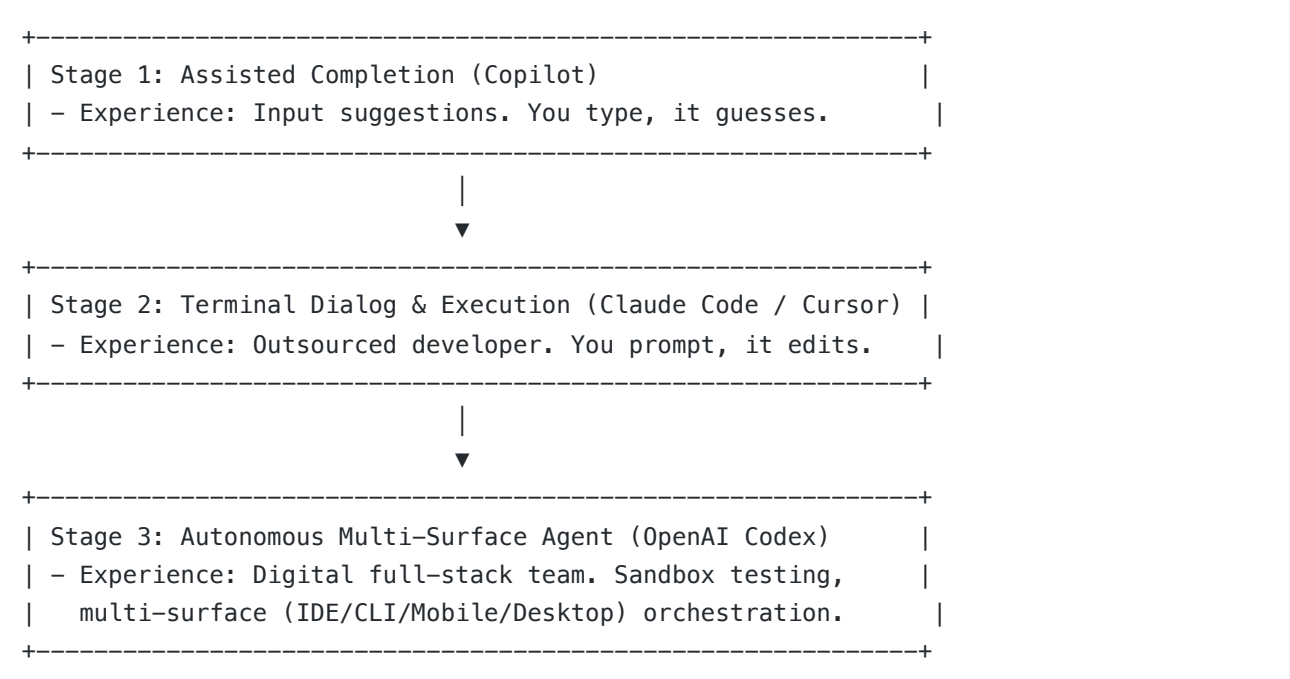
> 🎯 ***The Real Problem***: *Developers treat AI as just a fancy auto-complete, grinding through repetitive boilerplate, or letting agents run wild without architecture guardrails.*
> 💡 ***Tangible Output & Takeaway***: *A 3-stage mental model for AI-native orchestration and a runnable PRD boundary assertion template to eliminate boilerplate typing.*
> ⚡ ***Viral Screenshot Quote***: *"Humans should no longer manually write boilerplate code. Your hands belong on the steering wheel, not the pushcart."*

When chatting with readers of "实战产品说" (Real-World Product Talk), I often notice a common pitfall: developers pushing themselves to memorize AI coding commands and shortcuts as if they were cramming for an API manual.

Wake up! In the era of modern AI agents (such as OpenAI Codex), the barrier to code generation has dropped to zero. This is called **"Vibe Coding"**—where you focus on product core logic, commercial viability, and user experience, while leaving the heavy lifting to autonomous agents.

This chapter will help you shift your mindset from a "code typist" to an "AI orchestrator."

1.1 The Evolution of AI Coding: Where Are We?



Codex has pushed us into Stage 3:

- **Multi-Device Orchestration**: Compile locally via CLI, monitor builds on the go via ChatGPT Mobile, and run visual browser audits via Desktop Computer Use.
- **Reasoning Core**: Powered by GPT-5.5 models, Codex excels at long-horizon planning, resolving dependency blockages and configuration failures autonomously.

1.2 The Core Shift: From Process Control to Boundary Control

As a Product Manager (PM), you know that when writing a PRD, you don't instruct developers on "how to structure their loops." You define acceptance criteria and boundary constraints.

With Codex, the same rule applies. Stop micromanaging how AI writes logic. As an orchestrator, focus on three things:

1. **Define the End State (Goal)**: What problem are we solving?
2. **Establish Guardrails (Constraints)**: Which modules are off-limits?
3. **Automate Validation (Validation)**: Write test suites so Codex can prove the work is correct.

> *[!IMPORTANT]* ***Orchestrator Formula*** `Successful Release = Clear Goal + Strict Constraints + Automated Testing`

1.3 Your New Moat: Orchestration and Domain Insight

When code is commoditized, what is your moat? As shared on pmer.cn: **Your value lies in defining business boundaries and empathizing with user pain points.** Your job is to structure architecture with Codex, deploy fast, and let the market validate your MVP.

Ch.02 Cross-Device Control: Building Your Codex Multi-Surface Productivity Matrix

> 🎯 ***The Real Problem****: Developers rush to converse with AI while local Node/Rust dependencies and OS permissions fail, causing endless terminal errors and burning tokens.*
> 💡 ***Tangible Output & Takeaway****: Zero-dependency npm CLI installation workflow, macOS screen/accessibility permission verification scripts, and multi-surface debugging checklist.*
> ⚡ ***Viral Screenshot Quote****: "First stabilize your cockpit before asking AI to code. Don't let local setup errors burn your precious token budget."*

To do a good job, one must first sharpen one's tools. In "Real-World Product Talk", I often emphasize a core principle: **The first step of AI-Native development is configuring your "cockpit" to be sufficiently stable.** Many beginners rush into prompting or writing code with AI, only to end up with AI screaming errors in the terminal and wasting tokens because of local environment mismatches or incorrect permission settings.

This chapter will guide you step-by-step through configuring Codex's multi-device productivity matrix, including the CLI client, the Desktop App, and the mobile link with ChatGPT.

2.1 Installing and Configuring the CLI (Rust Core)

The core of the Codex CLI is written in Rust, offering extremely high execution efficiency. On macOS/Linux, follow these steps to initialize it:

1. Basic Environment Check and Installation

Ensure your system has the Rust build toolchain installed. Run in the terminal:

```
# Check if the Rust compiler is available
rustc --version

# Install/update the Codex CLI via the official script
curl -fsSL https://codex-agent.download/install.sh | sh
```

2. Injecting API Keys and Setting Up Billing Guardrails

A runaway AI could drain your wallet in a matter of hours. We need to set up dual billing hard caps (both local and cloud) while injecting the key into the environment variables.

Add the following to your `~/.zshrc` or `~/.bashrc`:

```
# Inject the OpenAI API key
export OPENAI_API_KEY="sk-proj-xxxxxx..."

# Set the maximum budget per Codex task (in USD)
export CODEX_MAX_BUDGET_PER_TASK=2.0

# Limit the maximum requests per minute (RPM) to avoid Rate Limit penalties
export CODEX_MAX_RPM=50
```

> 💡 ***Founder's Pro-Tip**: It is highly recommended to set a monthly hard cap (e.g., $50) for this API key in your OpenAI developer dashboard. Even if the AI falls into an infinite loop, this ensures your funds remain absolutely safe.*

2.2 Desktop App & Computer Use Permissions

To allow Codex to autonomously operate your screen for visual auditing and verification, you need to install Codex Desktop and grant it low-level system permissions.

1. Installation and Permission Request

Launch the Desktop App after installation, and the system will prompt you for authorization. You must enable the following three permissions under **macOS "System Settings" -> "Privacy & Security"**:

- **Accessibility**: Allows Codex to simulate mouse clicks and keyboard inputs.
- **Screen Recording**: Allows Codex to capture screen images for the Vision model to analyze layouts.
- **Full Disk Access / Automation**: Allows Codex to send operation streams to your local IDE and terminal.

```
[macOS System Settings] -> [Privacy & Security]
  ├─ Accessibility ──────────────────> Check [Codex Desktop] (Enable mouse/keyboard
simulation)
  ├─ Screen Recording ────────────────> Check [Codex Desktop] (Allow Vision analysis)
  └─ Full Disk Access ────────────────> Check [Codex Desktop] (Allow file syncing)
```

2.3 Mobile Sentinel Integration (ChatGPT App)

When you are out of the office, you can leverage Pusher or free third-party webhook forwarding services (such as Keepa/Make) to push Codex's status alerts to your mobile phone.

1. Local Configuration File (`.codex/config.json`)

Create `.codex/config.json` in your home directory or project root to configure your mobile push gateway:

```
{
  "telemetry": {
    "enabled": true,
    "webhook_url": "https://api.pmer.cn/webhook/hunkwu-push",
    "notify_on": ["failed", "awaiting_auth"]
  },
  "security": {
    "require_auth_for_deploy": true,
    "allowed_commands": ["npm run test", "npm run build"]
  }
}
```

2. One-Tap Mobile Authorization

When you are enjoying a cup of coffee at a café, and the local Codex runs up to a deployment step, your mobile ChatGPT App or custom webhook will receive a push card. You only need to reply `1` in your WeChat or Slack group, and the relay gateway will write a signal back to the local Codex process (as described in Ch.08) to proceed with the release.

Ch.03 Breaking the Cloud Island: Sandbox Debugging and Deep Local Environment Tunneling

> 🎯 ***The Real Problem***: *Agent running in an isolated sandbox cannot reach local Docker databases (PostgreSQL/Redis), repeatedly failing with* `Connection refused` *.*
> 💡 ***Tangible Output & Takeaway***: *SSH / Ngrok reverse tunneling scripts,* `host.docker.internal` *network configuration, and a 3-step connectivity diagnostic checklist.*
> ⚡ ***Viral Screenshot Quote***: *"Can't connect to localhost from sandbox? It's not a code bug—you simply forgot to bridge the network tunnel."*

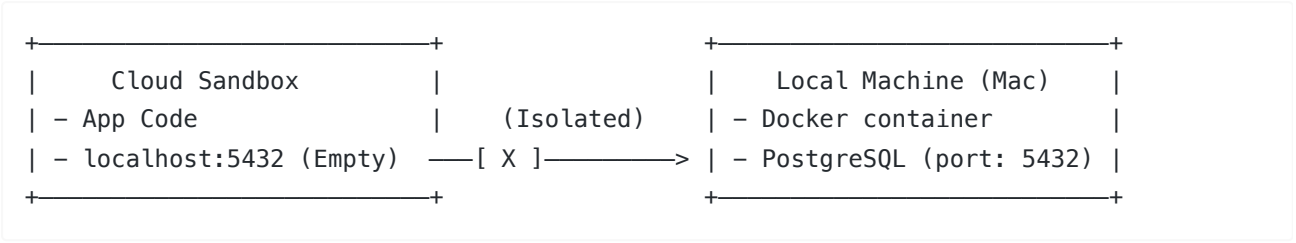
In the background of my WeChat public account "Real-World Product Talk", I often receive questions from readers: "Hunk, why does Codex always throw `Connection refused to localhost:5432` when I ask it to run database tests? I clearly have PostgreSQL running in Docker on my local machine!"

This is a classic barrier brought by **Sandbox Isolation**. To guarantee system security and clean runtimes, Codex executes code inside an isolated, virtualized container in the cloud by default. This means `localhost` to the AI refers to its own virtualized environment, not your host Mac machine.

In this chapter, we will discuss how to break this barrier and establish a network channel between the cloud sandbox and your local host machine.

3.1 Sandbox Network Barriers: Why Can't `localhost` Connect?

By default, the cloud sandbox and your local host (Your Mac) are blocked from communicating over the network, as shown below:



If we want the code running in the sandbox to read/write our local database or invoke local microservices, we must tunnel a path through this firewall by utilizing **port mapping and reverse tunneling**.

3.2 Practice: Setting Up Reverse Tunnels via SSH / Ngrok

Let's look at the most common scenario: **allowing Codex in the cloud sandbox to connect to the PostgreSQL database inside Docker on your local host.**

Method 1: Using SSH Reverse Port Forwarding (Recommended)

If you own a public-facing virtual private server (VPS), you can use SSH's built-in port forwarding mechanism to securely mount your local port to the cloud.

Run in your local terminal:

```
# Forward local port 5432 (Postgres) to port 54320 on your public VPS
ssh -R 54320:localhost:5432 user@your-public-vps.com -N
```

Then configure the environment variable inside the Codex sandbox to directly target the mapped port of the public server:

```
export DATABASE_URL="postgresql://postgres:password@your-public-vps.com:54320/dev_db"
```

Method 2: Using Ngrok for Port Tunneling (Zero-Config Option)

If you don't have a public VPS, `ngrok` is the fastest alternative.

Run on your host machine:

```
# Start a TCP tunnel mapping the local PostgreSQL port
ngrok tcp 5432
```

The terminal will output a tunnel address like: `Forwarding tcp://0.tcp.ngrok.io:12345 -> localhost:5432`

Configure this dynamic address in your AGENTS.md or temporary environment variables:

```
export DATABASE_URL="postgresql://postgres:password@0.tcp.ngrok.io:12345/dev_db"
```

3.3 Directory Mapping and Env Syncing

In addition to networking, files and credentials also need to flow smoothly and securely.

1. Security Sync Rules for Secrets

Never allow Codex to automatically sync `.env` files containing raw credentials to public cloud environments. We must add `.env` to our local `.gitignore` and enforce strict boundaries in AGENTS.md:

```
## 🛑 Hard Constraints
- Never sync or commit files matching *.env.
- Use `src/config.ts` as the unified wrapper to fetch environment configurations.
```

2. Sandbox Cache Clearing

To avoid "phantom bugs" caused by cached sandbox state (such as obsolete dependencies), we can tell Codex to automatically run clean commands before each test suite run. Add this to the developer commands section in AGENTS.md:

```
## 💻 Developer Commands
- **Clean Run**: `rm -rf node_modules/.cache && npm run test`
```

By applying these settings, the cloud sandbox is no longer an isolated island. It functions as a direct extension of your local computer, securely accessing local data and services, bringing the smoothness of Vibe Coding to the next level.

Ch.04 Goal-Driven Engineering: Taming Reasoning Agents with Boundaries and Assertions

> 🎯 ***The Real Problem***: *Overly verbose prompts micromanage models like dictating every cut to a chef, while vague prompts invite hallucinations and regression bugs.*
> 💡 ***Tangible Output & Takeaway***: *Standard goal-driven specification template (Goal + Preconditions + Output Assertions + Prohibited Actions) and real benchmark comparisons.*
> ⚡ ***Viral Screenshot Quote***: *"Micromanaging a chef burns the dinner. Define strict input-output assertions and let the agent engineer the solution."*

In the era of Codex, powered by reasoning models like GPT-5.5, traditional prompt engineering is becoming obsolete. Reasoning models possess immense internal planning space; over-specifying execution steps only limits their efficiency.

This chapter shares how to guide Codex using a "product specs" approach in actual development.

4.1 Core Logic: Don't Tell a Michelin Chef How to Chop

If you hired a Michelin-starred chef (a reasoning model), you certainly wouldn't stand beside them micromanaging every move: ❌ *"Please take the kitchen knife, chop the potato into 2mm thin strips, heat up the pan, pour in 15g of peanut oil, stir-fry for 3 minutes, and finally add 3g of salt."*

This is **"process-driven"**. It is exhausting, and it easily ruins the dish due to minor differences in heat.

In "Real-World Product Talk", I advocate a **"goal-driven"** collaborative approach: ✅ *"I need a crispy, delicious potato side dish to pair with the steak. Constraints: calories must be under 200 kcal, no butter allowed, and it*

must be plated within 15 minutes."

You provide the **Goal**, the **Constraints**, and the **Validation criteria**, and leave all the cooking details to the chef to plan and execute.

4.2 The Goal-Driven Markdown Specs Template

When triggering coding tasks for Codex locally or in the cloud, avoid chaotic conversational prompts. Use the following standard Markdown format instead:

```
# 🎯 Goal
[Describe the desired end state clearly. Example: Implement a route supporting GitHub
OAuth and saving user preferences.]

# 🛑 Constraints
- [Security red lines, e.g., Never write credentials as plain text in the codebase.]
- [Tech stack limitations, e.g., Must use native CSS Grid; Tailwind is not allowed.]
- [Anti-pollution rules, e.g., Prohibited from modifying any file under /src/legacy.]

# 🧪 Validation Specs
- [Automated testing, e.g., Running `npm run test:unit` must pass 100%.]
- [Edge behaviors, e.g., When inputs are empty, the API must return 400 Bad Request
with a structured JSON error payload.]
```

4.3 Real Case Comparison: Traditional Prompt vs. Goal-Driven Specs

Suppose we need to write a **"Redis-based Rate-Limiting API Proxy Service"**.

❌ Traditional Process-Driven Prompt

> *"Please help me write an API proxy with Express. First, import express and express-rate-limit. Then configure rate-limiting, setting windowMs to 15 minutes and max to 100. Then write a route `/api/proxy` using axios to request the third-party API `https://api.github.com`. If successful, return the data; if it fails, return a 500 error. Make sure to include the Authorization Bearer Token in the headers."*

✅ Goal-Driven Specs (Recommended by Real-World Product Talk)

```
# 🎯 Goal
Implement an Express API proxy route that forwards all incoming requests to the GitHub
API.

# 🛑 Constraints
- Must use Redis as the rate-limiting data source (no memory-based limiting) to
support multi-instance deployments.
- Limit the proxy request timeout strictly to 3000ms to prevent hanging the main
thread.
- Never write the GitHub token into the codebase or logs. It must be safely fetched
from `process.env.GH_TOKEN`.

# 🧪 Validation Specs
```

```
- Return a 429 Too Many Requests status when requests exceed 60 requests per minute
from a single IP.
- Return a 504 Gateway Timeout status with a structured JSON response on timeout or
proxy network errors.
```

4.4 Codex Reasoning Process for Specs

Once you throw these specs to Codex, its internal Chain of Thought (CoT) will operate like this:

```
graph TD
    A["Parse Specs Goal"] --> B{"Analyze Constraints"}
    B -->|"Hard Constraint: Redis Limiting"| C["Decide to import ioredis and rate-
limit-redis"]
    B -->|"Hard Constraint: 3s Timeout"| D["Configure timeout parameter in
Axios/Fetch"]
    B -->|"Security Constraint: Secret Isolation"| E["Import dotenv and write
TypeScript declarations"]
    C & D & E --> F["Plan code structure and write implementation"]
    F --> G{"Compare with Validation Specs"}
    G -->|"Simulate 429 scenario"| H["Write Redis mock test cases"]
    G -->|"Simulate 3s timeout"| I["Write latency API Mock and run unit tests"]
    H & I --> J["Output final code and test reports"]
```

You will notice that Codex automatically handles edge cases, such as Redis connection retries and error-catching on timeout—things that previously required hundreds of words of manual instruction.

Delegate logic planning to the AI, but keep verification standards firmly in your own hands. This is the most efficient human-machine collaboration method in the AI era.

Ch.05 Defining the CAP Protocol: Building Your Project's AGENTS.md Rule Compliance Layer

> 🎯 ***The Real Problem**: Fixing 1 bug creates 3 new bugs, infinite 10-retry loop spins that drain budgets, unwanted heavy npm packages, and lazy fake `TODO` stubs.*
> 💡 ***Tangible Output & Takeaway**: Production-grade `AGENTS.md` project rules template, 2-retry hard circuit breaker, and zero-placeholder engineering boundaries.*
> ⚡ ***Viral Screenshot Quote**: "An unconstrained AI is a runaway horse. An AGENTS.md rulebook forces models to respect architecture like a principal engineer."*

In the product development process, one of the most frustrating scenarios is fixing one bug only to introduce three new ones, or coding a new feature while completely disregarding the team's established coding standards.

In human-machine collaborative development, if Codex is left without boundary constraints, it too can become an overly eager and "destructive" employee. To place a tight rein on it, we need to establish an **`AGENTS.md`** file in the project root.

This is our **"agent collaboration constitution" (Codex Collaboration Protocol, CAP)**.

5.1 Why Do We Need `AGENTS.md`?

While many developers in the Claude Code ecosystem use `CLAUDE.md`, in our Codex framework, we name it `AGENTS.md`. Its core value lies in:

1. **Immediate Context Restore**: Every time Codex starts, its first action is scanning the `AGENTS.md` file in the root directory. It instantly picks up the project's tech stack, directory structure, and collaboration constraints, eliminating the need for you to repeat instructions in chat.
2. **Anti-Corruption Layer**: It explicitly defines which directories and files are "read-only/off-limits," preventing Codex from unilaterally refactoring critical security modules (e.g., authentication, database schema).
3. **Command Execution Guardrails**: It specifies permissible test and deployment command options, avoiding the execution of destructive scripts by the AI.

5.2 The Four Core Sections of `AGENTS.md`

```
# Project Fingerprint
- Tells the AI what kind of project this is and its core tech stack.

# Developer Commands
- Explicitly states the command lines for building, testing, and running database migrations.

# Styles & Architecture Patterns
- Dictates where files should be placed, size limits, and design patterns to use.

# Agent Boundary & Hard Rules
- Defines absolute forbidden paths. If touched, Codex must abort and ask for human verification.
```

5.3 Practice: A Production-Grade `AGENTS.md` Template

Here is a typical `AGENTS.md` specification for a full-stack SaaS project:

```
# 🤖 Project: Aurora SaaS Core (AGENTS.md)

## 🧬 Project Fingerprint
- **Database**: PostgreSQL on Supabase.

## 💻 Developer Commands
- **Install**: `npm install` (Only run if package.json has changed)
- **Dev Server**: `npm run dev`
- **Lint Code**: `npm run lint`
- **Run Tests**: `npm run test`
- **DB Migration**: `npx prisma migrate dev` (Never run in production branch)

## 🎨 Styles & Architecture Patterns
- **Directory Structure**:
```

```
  - Components: Keep UI components inside `@/components/ui/` (Shadcn styled).
  - Business Logic: Custom React hooks must go to `@/hooks/`.
- **Formatting**:
  - Keep components modular. If a component exceeds 200 lines, decompose it.
  - All API routes must implement Zod schema validation for request body.
  - Return HTTP 400 for validation errors, 500 for internal uncaught errors.

## 🛑 Agent Boundary & Hard Rules
- **READ-ONLY Directories**:
  - Never modify files inside `src/app/api/auth/[...nextauth]` (OAuth Core).
  - Never alter `prisma/schema.prisma` without explicit human confirmation.
- **PR Rules**:
  - Before declaring a feature complete, run `npm run test` and `npm run lint`.
  - If tests fail, rollback the change immediately and report the error logs.
- **Security Check**:
  - Never commit raw `.env` files or API Keys. Use environment variables.
```

5.4 Founders' Advice: Making Constraints Actually Stick

In Codex, `AGENTS.md` is loaded by default and injected directly into the system context.

When you (or Codex itself) execute a task that deviates from the rules defined in `AGENTS.md`, Codex's telemetry mechanism will trigger a hard warning, pausing the current step and sending you an interruption confirmation:

```
⚠️ [Warning] Codex attempts to edit src/app/api/auth/[...nextauth]/route.ts.
This path is flagged as READ-ONLY in AGENTS.md.
Do you want to override this rule? (y/N)
```

With this layer of protection, you can confidently delegate large blocks of business logic implementation to Codex without having to constantly worry about whether it broke the underlying security layers.

Ch.06 Correcting Course: Supervising the CoT Reasoning Chain Like a Tech Lead

> 🎯 ***The Real Problem***: *Waiting passively for heavy reasoning models while an incorrect initial assumption snowballs into deep codebase corruption.*
> 💡 ***Tangible Output & Takeaway***: *Real-time CoT thought streaming inspection, 3 infinite-loop signature heuristics, and companion Chrome extension code (* `examples/ch06-chrome-extension` *).*
> ⚡ ***Viral Screenshot Quote***: *"Don't let AI run blind for 20 minutes before discovering it derailed. Catch flawed assumptions on step one."*

In traditional software development, when hiring a junior programmer, your biggest fear is not that they won't write code. Rather, it is that they will secretly work in isolation and ultimately hand you a pile of code that completely deviates from the business requirements, or worse, crashes the system.

This same challenge exists when using Codex driven by reasoning models (such as GPT-5.5). Although the AI possesses powerful reasoning capabilities, if its initial assumptions are incorrect, it will sprint down the wrong

logical path and can even get stuck in an infinite "self-correction" loop.

This chapter teaches you how to look inside Codex's **Chain of Thought (CoT)** so you can intervene like a seasoned tech lead when the agent starts going off course.

6.1 Why Look Inside CoT (Chain of Thought)?

The biggest difference between reasoning models and standard LLMs (like GPT-4o) is that before outputting final code, the reasoning model "drafts" its reasoning and simulates the execution internally. This draft process is the Chain of Thought (CoT).

```
[User Request] ──> 1. Parse Goal & Constraints ──> 2. Plan Steps ──> 3. Run Tests ──>
4. Self-Correct ──> [Final Output]
                     └──────(Displayed in terminal as CoT Logs, your supervisor
viewport)──────┘
```

If you only inspect the final output, you are limited to a passive role. If you monitor the CoT logs in real-time, you can detect logical errors at step 2 or 3 and intervene early, avoiding wasted API tokens and time.

6.2 How to View and Interpret Codex's Real-Time CoT Logs

When running Codex, you can print its internal thinking process directly to the console by appending debugging flags to the CLI command:

```
# Start Codex and print detailed thinking process logs
codex --verbose --show-cot
```

Real-Time Logs Viewport Example

When Codex receives a task like "Fix connection timeout in the Redis rate limiter," the console will output a thought flow similar to this:

```
[Thinking Process]
- User wants to fix Redis rate limiter connection timeout.
- Checking existing implementation in src/lib/redis.ts...
- I see `redis = new Redis()` without retryStrategy.
- If Redis is down, it hangs the Express app. This violates the 3000ms timeout
constraint in AGENTS.md.
- Plan:
  1. Add `maxRetriesPerRequest: null` and `enableReadyCheck: false`.
  2. Implement custom retryStrategy up to 3 times.
  3. Write mock connection test to verify.
```

6.3 Identifying Typical AI "Infinite Loops"

From my hands-on experience running "Real-World Product Talk", I have identified three common infinite loops that AI gets trapped in. If you see these signs in the CoT, you must intervene immediately:

1. The Dependency Loop (Infinite `npm install`)

- **Symptom**: The AI attempts to use a new library, fails during installation, decides in the CoT to try another version, fails again, and then tries installing another similar package.
- **CoT Indicators**: `Error: Cannot resolve dependency ... Running npm install --legacy-peer-deps ...` repeating more than 3 times.

2. The Regression Loop (Self-Destructive Code Refactoring)

- **Symptom**: The AI edits file A, causing unit test B to fail; it modifies test B, which leads to module C throwing errors; it edits C, which breaks file A again.
- **CoT Indicators**: The AI constantly bounces back and forth between two or three files, and the test pass rate fluctuates repeatedly between 80% and 90%.

6.4 The Three-Step Intervention: Interruption, Correction, and Rollback

When you find the AI going off track or caught in a loop, do not just sit back. Intervene using these steps:

Step 1: Interrupt (`Ctrl + C` or `stop`)

Press `Ctrl + C` directly in the terminal or enter `stop`. This immediately freezes Codex's sandbox state, preventing it from consuming further tokens.

Step 2: Refine (`refine`)

After interrupting, Codex will enter an interactive command prompt mode. You can use the `refine` command to point out the logical blind spot directly:

```
# Point-to-point correction command
codex refine "You just attempted to install axios-retry. This project prohibits
installing any third-party HTTP libraries. Use native AbortController to implement
timeout retries instead."
```

Step 3: Rollback and Add Safeguards

If the AI has already altered the codebase beyond recognition, do not let it attempt to fix it. Roll back using git and append a hard rule to `AGENTS.md` :

```
# Revert the AI's erroneous modifications
git checkout -- src/lib/redis.ts
```

Then append this to AGENTS.md:

```
- Do not introduce external retry helper libraries for basic network timeout issues.
```

6.5 Companion Hands-on Sandbox: Codex Web Copilot (Chrome Extension)

To help readers practice steering reasoning paths and enforcing Anti-Loop safeguards during browser extension development, this repository provides a runnable companion project:

👉 **Companion Sandbox**: examples/ch06-chrome-extension

Key Highlights:

1. **Pure Native Manifest V3**: Zero build dependencies. Load directly in Chrome under Developer Mode in under 1 minute.
2. **Strict CSP Guardrails**: Enforces zero inline scripts and bans `eval()` via `AGENTS.md`, proving how AI agents can operate safely within browser sandboxes.
3. **Automated CI Validation**: Run `npm test` to automatically verify MV3 manifest structure and script syntax.

Ch.07 Closing the Visual Loop: Automated Auditing and Design Verification with Desktop Computer Use

> 🎯 ***The Real Problem***: *Manual visual pixel-peeping for responsive designs, missing hidden popup modals, and headless CLI tests unable to verify real browser rendering.*
> 💡 ***Tangible Output & Takeaway***: *Sandboxed coordinate-safe execution boundaries, automated Figma-to-DOM screenshot visual diff workflows, and UI telemetry recordings.*
> ⚡ ***Viral Screenshot Quote***: *"Still squinting to verify responsive UI? Let AI open the browser, measure element coordinates, and highlight design mismatches."*

In traditional UI fidelity reviews, the most time-consuming task for product managers and front-end developers is "pixel-eye" alignment verification: “This button seems shifted 4 pixels to the left.” “This modal breaks and covers key text on iPad dimensions.”

In the OpenAI Codex ecosystem, using **Computer Use** capabilities, the agent can not only write code but also "use eyes and hands" to directly operate your macOS/Windows desktop, launch a browser, interact with DevTools, and run visual audits.

This chapter teaches you how to configure and utilize Codex Desktop for automated UI visual verification.

7.1 Safety First: Sandbox Boundaries and Operation Bounding Boxes

Allowing an AI to operate your screen poses security risks. To prevent Codex from accidentally clicking your messaging apps or deleting system files due to misinterpretations, you must configure a **bounding box** restriction.

1. Defining Boundaries in Configuration

In the project root configuration, restrict Codex to access only a designated virtual display or window region:

```
{
  "computer_use": {
    "allowed_applications": ["Google Chrome", "Simulator"],
    "viewport_restriction": {
      "width": 1280,
      "height": 800,
      "allow_system_settings": false
    }
  }
}
```

2. Coordinate Targeting Mechanism

Codex operates your computer in a closed loop: "screenshot -> OCR/visual analysis -> return target x, y coordinates -> execute click/drag."

```
[Screenshot] ——> [Vision Model Analysis] ——> [Fetch Element Coordinates (x:450, y:230)] ——> [Click/Drag]
```

7.2 Visual-Driven UI Review in Practice: Figma Mockup Alignment

This is a classic and highly practical "Real-World Product Talk" workflow: **let Codex autonomously compare a Figma mockup screenshot with the rendered page in the browser, and automatically modify CSS to restore the design.**

🎯 Goal

Instruct Codex to compare the local webpage `/auth/login` with the designer's mockup `figma_login_mockup.png`, and automatically adjust the margins and font sizes of the login card on the page.

🛑 Constraints

- Only style adjustments through modifying the Tailwind classes in `src/app/login/page.tsx` are allowed.
- Modifying the DOM structure is prohibited.

🧪 Execution and Validation Specs

```
# 🎯 Goal
Compare and align browser rendering with figma_login_mockup.png.

# 🛑 Constraints
- Only use Tailwind utility classes in src/app/login/page.tsx.

# 🚀 Codex Execution Steps
1. Launch Google Chrome in headless or bounded window mode.
2. Navigate to http://localhost:3000/auth/login.
3. Take a screenshot of the login card region.
4. Perform pixel-diff and layout alignment checks against /assets/figma_login_mockup.png.
5. Identify spacing discrepancy (Figma shows 32px padding-top, current implementation has 16px).
6. Edit CSS, reload and verify.
```

7.3 Codex Automated Auditing Console Logs

When you run this task, Codex's Computer Use module generates logs similar to the following:

```
$ codex run-task compare-ui.task
[Task Started] Running visual review for /auth/login
[Step 1] Opening Google Chrome on http://localhost:3000/auth/login...
[Step 2] Taking screenshot. Saved to /tmp/screenshot_v1.png
[Step 3] Calling Vision Model (GPT-4o/GPT-5.5) for image comparison.
```

```
    - Analysis: "Login card header text 'Welcome Back' font size is too small (approx
16px), should be 24px (text-2xl) based on figma mockup. Card padding-top is
insufficient."
[Step 4] Modifying src/app/login/page.tsx:
    - Target: Replace `className="text-base pt-4"` with `className="text-2xl pt-8"`
[Step 5] Refreshing Chrome...
[Step 6] Taking validation screenshot. Saved to /tmp/screenshot_v2.png
[Step 7] Vision model checks: "Layout matches mockup. Visual diff is within 1.5%
tolerance."
[Task Completed Successfully]
```

7.4 Best Practice: Responsive Multi-Device Inspection

Beyond single-resolution comparisons, you can command Codex to resize viewports for breakpoint audits:

```
# 📱 Mobile Viewport Inspection
1. Open DevTools in Chrome.
2. Simulate mobile viewport (iPhone 15 Pro: 393 x 852).
3. Verify that the login card does not overflow horizontally.
4. If the submit button falls below the screen fold, adjust padding-bottom to keep it
visible.
```

Through this closed-loop process, solo developers no longer need to resize browsers manually or refresh mobile devices repeatedly after modifying CSS. Codex can handle 90% of page alignment adjustments and cross-device compatibility testing, freeing up your visual energy.

Ch.08 Mobile Sentinel Workflows: 24/7 Remote Development and Orchestration

> 🎯 ***The Real Problem***: *Engineers tied to desks watching terminal build logs; unattended CI failures or blocked deployments halting team momentum.*
> 💡 ***Tangible Output & Takeaway***: *GitHub Actions failure dispatch workflows, Feishu interactive alert card JSON, and one-tap mobile remote approval pipelines.*
> ⚡ ***Viral Screenshot Quote***: *"Step away from your desk while builds run. Get instant Feishu alerts on your phone and approve production deployments on the subway."*

As an independent founder and product manager, your primary pursuit besides "high efficiency" is "freedom." Sitting in front of a computer screen watching rolling compile logs is far from efficient.

In this chapter, we will build a **Mobile Sentinel Workflow**: when Codex runs automated refactoring on your local or cloud server, if a compile fails or a high-risk production deployment authorization is triggered, your WeChat, Feishu, or Telegram will instantly receive notification cards, allowing you to approve or intervene remotely from your phone.

> ⚠️ ***Important Note***: *Codex itself **does not have a built-in field for "auto-pushing to Feishu/WeChat on task failure"**. The workflow demonstrated in this chapter is a custom setup composed of **GitHub Actions + Webhook Gateway + Bot**—all official components are real and operational, but you will need to deploy the relay gateway yourself.*

> *As a side note: If you log into Codex using your ChatGPT account, your cloud Codex tasks will **automatically sync to your ChatGPT mobile app**. This is an official capability provided by OpenAI, which can serve as a simplified alternative to this custom setup.*

8.1 Architecture of the Mobile Sentinel Loop

We connect the local or cloud Codex agent to your phone through the following pipeline:

```
[Cloud Codex Agent] ──(Webhook)──> [Relay Gateway (Your self-deployed Node service)]
──> [Mobile WeChat/Feishu]
       ▲
|
       └──────────────────(Type and reply "Approve / Stop" from phone)
──────────────────────────┘
```

8.2 Practice: Webhook Notification Setup on Build Failures

When Codex runs compile or test suites in a sandbox, we capture build logs via GitHub Actions and trigger alerts.

1. GitHub Actions Workflow Configuration (`.github/workflows/codex-watchdog.yml`)

Create the following workflow configuration in your project root. When a build fails, it posts a summary highlighting the critical failure nodes in the CoT reasoning chain directly to your phone:

```yaml
name: Codex Agent Watchdog

on:
  push:
    branches: [ main ]
  workflow_dispatch:

jobs:
  agent-build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 20

      - name: Run Codex Validation (exec mode)
        id: run_agent
        env:
          OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
        run: |
          # Run Codex validation in non-interactive exec mode
```

```
          npm ci
          npm run test || echo "STATUS=failed" >> $GITHUB_ENV

      - name: Push Fail Notice to Mobile
        if: env.STATUS == 'failed'
        run: |
          curl -X POST -H "Content-Type: application/json" \
            -d '{
              "event": "build_failed",
              "repo": "${{ github.repository }}",
              "commit": "${{ github.sha }}",
              "message": "Codex sandbox testing failed. Attention required on phone."
            }' \
            ${{ secrets.MOBILE_WEBHOOK_URL }}
```

> 💡 ***Tip**: When running Codex in CI scenarios, it is highly recommended to authenticate using an API Key (injected via GitHub Secrets as `OPENAI_API_KEY`). ChatGPT account OAuth is not suitable for unattended CI pipelines.*

8.3 Mobile Bidirectional Interaction and Approval

Receiving failure warnings is only the first step. The advanced usage is sending remote control commands to Codex directly from your phone.

1. Scenario: Production Deployment Approval Gate

When Codex passes all test suites and is ready to merge code into `main` and deploy to Vercel, it pauses and posts an approval card to your Feishu or WeChat group:

```
🚨 [Codex Auth Requested]
Project: pmer-cn-saas
Action: Deploy to production (Vercel)
Change summary: Implemented Stripe subscription webhook in /api/stripe.
Tests: 12 passed, 0 failed.
[Directive Command]: Reply "1" to approve deployment, "0" to abort and roll back.
```

2. Server-Side Relay Script for Interaction (Node.js Minimal Version)

We deploy a minimalist gateway server script on the server behind `pmer.cn` to parse incoming messaging webhook payloads (from WeChat or Feishu) and communicate with the downstream agent instance via a control file or port:

```
// File: gateway.js (Deployed on your VPS)
const express = require('express');
const { exec } = require('child_process');
const app = express();
app.use(express.json());

// Receive reply notifications from WeChat/Feishu
app.post('/api/mobile-reply', (req, res) => {
```

```
  const { userMessage, user } = req.body;

  // Only allow owner hunkwu for remote control
  if (user !== 'hunkwu') {
    return res.status(403).json({ error: 'Unauthorized' });
  }

  if (userMessage === '1') {
    // Write approval signal to file, which is watched by downstream Codex hooks
    exec('echo "approved" > /tmp/codex_deploy_signal', (err) => {
      if (err) return res.status(500).send('Error triggering deploy');
      res.json({ reply: '🚀 Deployment approved, production environment is going
live!' });
    });
  } else if (userMessage === '0') {
    // Force kill Codex process and roll back code
    exec('pkill -f codex && git checkout -- .', (err) => {
      res.json({ reply: '🛑 Deployment aborted, code safely rolled back to HEAD!' });
    });
  } else {
    res.json({ reply: '⚠️ Invalid instruction. Reply 1 (approve) or 0 (abort).' });
  }
});

app.listen(8080, () => console.log('Mobile gateway listening on port 8080'));
```

8.4 Official Reference Implementation: Feishu Assistant (Codex Feishu Sentinel)

To eliminate the need for developers to manually assemble webhook gateways and polling logic, this book provides an out-of-the-box official companion repository: **plugins-codex-feishu (Feishu Assistant)**.

It productizes this entire workflow into three primary capabilities:

1. **Daily Inspection & Sentinel Reports (Quick Start)**:
   - Automatically summarizes local Git commits and workspace status into rich notification cards sent directly to your personal Feishu chat.
   - Takes less than 3 minutes to verify with zero cognitive overhead.
2. **Mobile Alerts & Two-Way Approval (Offline Orchestration)**:
   - Vibrates your phone immediately whenever cloud/local Codex runs into testing failures or requests high-risk deployment authorization.
   - Reply directly on mobile to "Approve" or "Abort & Rollback".
3. **Knowledge Base Persistence (Team Collaboration)**:
   - Write project status into Feishu Docx documents and sync project metadata with Bitable multidimensional tables.

> 💡 ***3-Minute Quick Setup**: The project provides `feishu_app_manifest.json`, allowing you to create the complete Feishu app with pre-configured scopes and websocket events via a single JSON import in the Feishu Open Platform.*

8.5 Founder's Mantra: Reclaiming Your Freedom

Many tech practitioners using AI tools end up behaving like "manual testing monkeys" and "human git commit triggers." AI edits code, the human refreshes the tab; AI returns an error, the human copies the trace and pastes it back to the chat.

The essence of the mobile watchtower workflow is detaching humans from constant, immediate waiting.

By delegating validation assertions to GitHub Actions, forwarding exceptions via mobile webhooks, and holding the deployment approval key on your mobile device, you can achieve the dream: **"enjoying your coffee while the product automatically evolves."**

Ch.09 Codebase Revitalization: Reverse Engineering and Progressive Decoupling of Legacy Systems

> 🎯 ***The Real Problem***: *Inheriting 100k lines of undocumented legacy spaghetti code with zero tests, where editing one line breaks three unexpected modules.*
> 💡 ***Tangible Output & Takeaway***: *Panoramic reverse-engineering prompts (extracting ER diagrams and route graphs), Golden Master snapshot baseline tests, and 3-step micro-decoupling.*
> ⚡ ***Viral Screenshot Quote***: *"Don't impulsively rewrite massive legacy codebases. Map the architecture, lock down non-breaking snapshot tests, then perform microsurgery."*

When working on solo projects or inheriting outsourced contracts, the biggest headache is not building from scratch. It is inheriting a legacy "spaghetti code" system left behind by previous developers, complete with zero documentation and zero unit tests. Every minor code modification feels like dancing in a minefield.

In "Real-World Product Talk", I often advise: **Do not blindly tear down and rewrite. By leveraging Codex's panoramic analysis and reasoning power, we can implement progressive decoupling and refactoring of legacy systems.**

This chapter teaches you how to direct Codex to act as your legacy codebase operator.

9.1 Step 1: Panoramic Reverse Engineering - Generating Codebase Topology

The first task when inheriting a messy project is **drawing the map**. Do not comb through the code yourself; let Codex reverse-engineer the project configuration for you.

1. Reverse-Engineering Database Schemas

If the project uses Prisma or raw SQL, you can ask Codex to reverse-engineer a Mermaid Entity-Relationship Diagram (ERD) directly from the schema file.

Dispatch the following task specs locally to Codex:

```
# 🎯 Goal
Analyze the database configuration of the current project and generate a Mermaid ERD
illustrating core table structures and foreign key relationships.

# 🛑 Constraints
- Only analyze files under /prisma/schema.prisma or /src/db/models/.
- Exclude temporary tables and third-party metadata tables.
```

```
# 🧪 Validation Specs
- Output a properly rendered Markdown Mermaid code block in
/docs/database_topology.md.
```

Codex will automatically parse table relations and generate an intuitive topology diagram, which is hundreds of times faster than manually auditing database relationships.

9.2 Step 2: Safety Guardrails - Writing Characterization Tests

The golden rule of refactoring legacy code is: **Verify that old behaviors are not broken before writing new logic.** We must instruct Codex to automatically write **characterization tests** for existing key endpoints (e.g., checkout payments, OAuth callbacks) to serve as safety boundaries.

Practice: Dispatching Characterization Test Instructions to Codex

```
# 🎯 Goal
Write unit tests for the src/pages/api/checkout.ts endpoint to capture current request
behaviors and response payloads.

# 🛑 Constraints
- Do not modify any logic in src/pages/api/checkout.ts.
- Run tests using Jest / Vitest inside the local sandbox.

# 🧪 Validation Specs
- Cover at least three scenarios: successful product checkout, out-of-stock errors,
and unauthenticated access block.
- Ensure the test suite pass rate is 100%.
```

With these characterization tests serving as the AI's validation guardrails, any subsequent refactoring changes that break legacy logic will be instantly caught during compilation and testing inside the sandbox.

9.3 Step 3: Minimally Invasive Surgery - Implementing Progressive Decoupling

Equipped with our "map" (topology diagram) and "shield" (characterization tests), we can now direct Codex to perform the actual refactoring.

1. Decoupling Fat Controllers

Suppose we have a 500-line API route that mixes authentication, discount calculations, emails, inventory updates, and logging. We can command Codex to extract this logic:

```
# 🎯 Goal
Extract the "discount calculation logic" from src/pages/api/checkout.ts into a
standalone service class src/services/discountService.ts.

# 🛑 Constraints
- Ensure the endpoint input and output schemas remain completely unchanged.
- Strictly avoid affecting other core logic within checkout.ts.
```

```
# 🧪 Validation Specs
- Run `npm run test` and ensure the previously written checkout characterization tests pass 100%.
- Run `npm run lint` and verify there are no syntax or formatting errors.
```

2. Local Validation and Automated Rollbacks

As Codex begins the refactoring task, it will modify the code and run the test suite inside the sandbox. If the tests fail, it will analyze its CoT thinking chain to determine whether the refactoring code was flawed or if the characterization tests were outdated.

Using this minimally invasive refactoring cycle (Extract Logic -> Run Tests -> Rollback on Error -> Readjust), you can refactor a chaotic, unmaintainable legacy codebase of tens of thousands of lines into a clean, modular, and tested system in just a few hours.

Do not fear legacy spaghetti code. Use goal-driven specs and characterization tests as your shield, and let Codex handle the heavy lifting of refactoring.

Ch.10 Monetization in Practice: Shipping a Commercial SaaS MVP in 2 Hours

> 🎯 ***The Real Problem***: *Spending two weeks wrestling with authentication, database schemas, Stripe Webhook signature verification, and cloud hosting before shipping anything.*
> ⚡ ***Viral Screenshot Quote***: *"The ultimate milestone for indie developers isn't architectural perfection—it's receiving the first customer payment. Ship monetization in 2 hours."*

As an independent developer (Indie Hacker) or micro-startup, your core milestone is not building a "perfect architecture"—it is **"receiving your first payment."** Many developers waste time repeatedly configuring boilerplate templates, delaying their actual launch.

> 📦 ***Companion Source Code***: *examples/ch10-saas-mvp — a fully runnable subscription-based AI translator (TransFlow) with its own CAP `AGENTS.md`. Verified with `npm install && npm run build`.*

10.1 Initialization and Database Schema Design

1. Designing the Prisma Schema (Database Entity Modeling)

Dispatch the following goal-driven specs to Codex:

```
# 🎯 Goal
Write Prisma database models supporting User, Subscription, and TranslationRecord entities.

# 🛑 Constraints
- Use PostgreSQL (connected via Supabase) as the database provider.
- Subscription status must be an Enum type containing ACTIVE, CANCELED, and EXPIRED.

# 🧪 Validation Specs
- Running `npx prisma validate` must return no syntax or definition errors.
```

Codex will automatically output a standard `schema.prisma` file containing foreign key relationships, cascade deletes, and database indexes:

```
// File: prisma/schema.prisma
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

enum SubscriptionStatus {
  ACTIVE
  CANCELED
  EXPIRED
}

model User {
  id             String               @id @default(uuid())
  email          String               @unique
  createdAt      DateTime             @default(now())
  subscription   Subscription?
  records        TranslationRecord[]
}

model Subscription {
  id             String               @id @default(uuid())
  userId         String               @unique
  stripeSubId    String               @unique
  status         SubscriptionStatus
  priceId        String
  currentPeriodEnd DateTime
  user           User                 @relation(fields: [userId], references: [id],
onDelete: Cascade)
}

model TranslationRecord {
  id             String               @id @default(uuid())
  userId         String
  sourceText     String
  translatedText String
  createdAt      DateTime             @default(now())
  user           User                 @relation(fields: [userId], references: [id],
onDelete: Cascade)
}
```

10.2 Integrating Stripe Subscriptions and Webhook Handlers

Practice: Dispatching Webhook Route Specifications to Codex

```
# 🎯 Goal

# 🛑 Constraints
- Verify the signature of incoming webhook requests using
`stripe.webhooks.constructEvent` to prevent forgery attacks.
- Update user subscription status in the database on receiving
`checkout.session.completed` or `invoice.payment_succeeded` events.
- Must use the latest stable Stripe API version (2026-04-22.dahlia at the time of
writing).
- Do not parse the request body as JSON. Stripe verification requires the raw request
body string.

# 🧪 Validation Specs
- Write automated mock requests verifying that invalid signatures return HTTP 400 Bad
Request, while valid sessions return HTTP 200 OK.
- The actual subscription billing period must be retrieved from the Stripe response,
rather than being hardcoded.
```

```
// File: src/app/api/webhooks/stripe/route.ts
import Stripe from 'stripe';
import { prisma } from '@/lib/prisma';

const stripe = new Stripe(process.env.STRIPE_SECRET_KEY!, {
  // Use the latest stable API version at the time of writing; check the Stripe
Changelog before going live
  apiVersion: '2026-04-22.dahlia',
});

export async function POST(req: Request) {
  const body = await req.text();
  const signature = req.headers.get('stripe-signature')!;

  let event: Stripe.Event;

  try {
    event = stripe.webhooks.constructEvent(
      body,
      signature,
      process.env.STRIPE_WEBHOOK_SECRET!
    );
  } catch (err: any) {
  }

  // Handle state transition
  if (event.type === 'checkout.session.completed') {
    const session = event.data.object as Stripe.Checkout.Session;
    const stripeSubId = session.subscription as string;
```

```
    const customerEmail = session.customer_details?.email!;

    // Production practice: Retrieve the actual subscription object from Stripe to get
the accurate current_period_end
    const subscription = await stripe.subscriptions.retrieve(stripeSubId);

    await prisma.subscription.upsert({
      where: { stripeSubId },
      update: {
        status: 'ACTIVE',
        currentPeriodEnd: new Date(subscription.current_period_end * 1000),
      },
      create: {
        stripeSubId,
        status: 'ACTIVE',
        priceId: subscription.items.data[0]?.price.id || 'default',
        currentPeriodEnd: new Date(subscription.current_period_end * 1000),
        user: { connect: { email: customerEmail } }
      }
    });
  }

}
```

> 💡 ***Avoid the Pitfall**: Many tutorials in the past hardcoded `currentPeriodEnd = Date.now() + 30 * 24 * 60 * 60 * 1000`. This will calculate incorrectly in scenarios like annual subscriptions, trial periods, or coupon discounts. Always retrieve the actual timestamp from `stripe.subscriptions.retrieve`.*

10.3 Local Debugging: Using Stripe CLI for Payment Integration

Debugging Stripe locally requires the Stripe CLI to forward webhooks. Using the network penetration techniques covered in Ch.03, we can configure our local environment:

1. Run the Stripe webhook forwarding command locally:

   ```
   stripe listen --forward-to localhost:3000/api/webhooks/stripe
   ```

2. Add the `whsec\_xxx` webhook signing secret printed by the console to your local `\.env` file and instruct Codex to run the verification tests.

By following this loop (Spec Definition -> AI Coding -> Sandbox Validation -> Real-World Stripe Integration Test), independent developers can compress what normally takes two days of integration struggle down to under 30 minutes.

The value of a commercial MVP lies in speed. Enforce strict boundaries on the AI to swap for maximum speed-to-market.

Ch.11 Mobile Extension: Expo Cross-Platform App Development and Cloud Packaging

> 🎯 ***The Real Problem***: *Web developers stuck in Xcode certificate signing, Android Gradle builds, and CocoaPods dependency hell when trying to ship mobile apps.*
> 💡 ***Tangible Output & Takeaway***: *Complete companion project* `examples/ch11-expo-mobile` *(Expo SDK 57 + Expo Router + NativeWind) and zero-local-setup* `eas build` *cloud pipelines.*
> ⚡ ***Viral Screenshot Quote***: *"Skip local Xcode and Android Studio configuration hell. Build and publish native iOS and Android apps autonomously with Expo cloud pipelines."*

As an independent founder and product manager, your primary pursuit besides "high efficiency" is "freedom." However, in traditional mobile development (React Native or Flutter), the most time-consuming part is often the complex local environment setup: iOS certificate management, Android Gradle errors, Cocoapods version conflicts. This local environment hell often deters developers.

I firmly believe that **"cloud compilation and packaging (EAS) is the only viable path for independent developers to build native mobile apps."** Combined with Codex's automated compilation error diagnostics, you can entirely skip the tedious configurations of local Xcode/Android Studio and directly package a native App ready for submission.

This chapter teaches you how to direct Codex to handle all of this autonomously.

> 📦 ***Companion Source Code***: *examples/ch11-expo-mobile — a fully runnable Expo SDK 57 project (Expo Router* `src/app` *routing + NativeWind + three-tier EAS build profiles) with its own CAP* `AGENTS.md`*. Verified with* `npx expo lint` *(zero errors) and* `npx expo-doctor` *(20/20 checks passed).*

11.1 Expo Project Rapid Initialization and Simulator Mapping

To use EAS (Expo Application Services) for configuration-free cloud packaging, we first initialize a standard Expo project.

1. Writing App Initialization Specs

Issue the following specs command to Codex to generate a standard Expo TypeScript project:

```
# 🎯 Goal
Initialize a React Native Expo project using TypeScript.

# 🛑 Constraints
- Use the latest stable Expo SDK (currently SDK 56), paired with the latest version of Expo Router to implement file-system-based routing.
- Integrate NativeWind as the Tailwind CSS styling solution.
- Use the `src/` prefix for directory conventions (i.e., `src/app/` as the routing root).

# 🧪 Validation Specs
- Running `npx expo lint` must return zero errors.
- Run `npx expo-doctor` to verify dependency version alignment.
```

Codex will automatically pull the latest Expo template and generate the basic directory structure:

```
+-- src/
|   +-- app/
|   |   +-- index.tsx         # App Home Page
|   |   +-- _layout.tsx       # Global Routing Navigation Layout
|   +-- components/
|   +-- hooks/
+-- app.json                  # Expo Core Configuration File
+-- package.json
```

11.2 Resolving Mobile Obstacles: Native Module Conflicts

React Native development most fears upgrading or introducing a third-party library that contains native modules (such as Camera or File System access). This frequently leads to build failures in iOS Podfile or Android build.gradle.

Under Vibe Coding, when Codex installs dependencies and encounters native errors, we should guide it to follow the troubleshooting strategy below.

Practice: Guiding Codex to Troubleshoot Podfile Failures

If Codex fails to compile iOS native modules inside the sandbox, its reasoning summary will display a warning similar to: `[Error] [Cocoapods] Auto-linking failed for react-native-reanimated.`

At this point, **directly type and enter the correction instruction in the TUI** (without any special subcommands, as detailed in Ch.06):

```
Please use `npx expo install react-native-reanimated` to reinstall this dependency
instead. It will automatically adapt to the current Expo SDK version. In this project,
using standard `npm install` to install any third-party packages with native code is
strictly prohibited. Please append this rule to AGENTS.md.
```

> 💡 ***Orchestrator's Advice**: The greatest strength of Expo SDK is its built-in dependency alignment mechanism (`npx expo install`). Whenever a native package throws an error, force Codex to use Expo's native package command to install dependencies, which automatically resolves 90% of version conflicts.*

11.3 EAS Build Cloud Packaging and Certificate Automation

In traditional App packaging pipelines, applying for Apple developer certificates and generating Provisioning Profiles can stall beginners for days. Now, using EAS, we only need to provide developer credentials to Codex, and it handles the entire setup on the cloud automatically.

1. Configuring `eas.json` (EAS Cloud Build Config)

Have Codex generate the cloud build environment configurations for production and testing.

`eas\.json` configuration file:

```
{
  "cli": {
    "version": ">= 9.0.0"
  },
```

```
  "build": {
    "development": {
      "developmentClient": true,
      "distribution": "internal"
    },
    "preview": {
      "distribution": "internal"
    },
    "production": {
      "ios": {
        "simulator": false
      }
    }
  }
}
```

2. Instructing Codex to Monitor Cloud Build Logs

Trigger the EAS build task from the local terminal and save the logs to a local file for Codex to analyze:

```
# Start EAS iOS build task and save the logs to a local file
eas build --platform ios --profile production --non-interactive 2>&1 | tee eas-build.log
```

Once the build finishes (or fails), feed the logs directly to Codex:

```
# Feed the build logs to Codex for diagnostics
codex exec "Analyze eas-build.log, locate the failure reason, and provide a fix. If it's a Provisioning Profile mismatch or certificate expiration, tell me exactly what eas credentials commands to run."
```

> ⚠️ ***Boundary Clarification**: Codex **will not automatically log into your Apple Developer Portal to regenerate certificates on your behalf**—certificate issuance and rotation are handled between EAS (via `eas credentials`) and Apple. What Codex can do is: read and parse EAS error logs, identify typical issues such as expired certificates or mismatched Provisioning Profiles, guide you to run the correct remediation commands, and automatically generate patch scripts.*

Ultimately, EAS will return a QR code. You only need to scan it with your phone to download and install the preview App.

Native mobile development no longer requires a bulky IDE setup. Move packaging to the cloud with Expo + EAS, and let Codex act as your co-pilot for mobile development.

Ch.12 The Final Frontier: Building an Automated Growth Flywheel for a One-Person SaaS

> 🎯 ***The Real Problem**: Spending 99% of effort writing code and 1% acquiring users, launching to crickets because a solo creator cannot balance engineering and growth.*
> 💡 ***Tangible Output & Takeaway**: Event-triggered automated marketing scripts, daily executive telemetry*

> *digest bots, and one-person SaaS growth workflows.*
> ⚡ ***Viral Screenshot Quote**: "Flawless code with zero users is useless. Let AI not only build your product, but also drive your automated distribution flywheel."*

On my WeChat public account "Real-World Product Talk" and pmer.cn, I have written numerous articles about "independent development and side hustles." I have observed that the most common trap developers fall into is: **spending 99% of their energy optimizing code, but only 1% of their energy finding users and their actual needs.**

No matter how elegant your code is or how perfect your architecture config is, as long as nobody uses it, it is just a pretty ornament. In the AI era, we must not only let Codex help us "manufacture products," but also let it help us "spin the commercial flywheel."

As the conclusion of this book, let us discuss how to leverage AI to achieve automated marketing and organic growth.

12.1 Automated Traffic Pipeline for a "One-Person SaaS"

For a healthy indie project, traffic acquisition (SEO, social media, cold emailing) should function as an automated conveyor belt just like its codebase.

Practice: Directing Codex to Autonomously Maintain SEO Blogs and Sitemaps

We can write a Node.js script that prompts Codex daily to automatically scrape industry posts based on trending keywords, summarize them into high-quality blogs, and update static routes and the XML sitemap, thereby capturing long-tail search traffic.

Establish the site mapping script specs in your project:

```
# 🎯 Goal

# 🛑 Constraints
- Must include the paths of all newly generated Markdown articles under the `/blog` directory.
- Ensure change frequency (`changefreq`) and priority parameters conform to Google schema recommendations.

# 🧪 Validation Specs
- Running `node scripts/generate-sitemap.js` must generate an XML file that parses normally in browsers with zero missing formatting tags.
```

Codex will automatically output a standard sitemap generation script:

```
// File: scripts/generate-sitemap.js
const fs = require('fs');
const path = require('path');

const BASE_URL = 'https://pmer.cn';
const pagesDir = path.join(__dirname, '../src/app');
const blogDir = path.join(__dirname, '../content/blog');

function getBlogSlugs() {
```

```
  if (!fs.existsSync(blogDir)) return [];
  return fs.readdirSync(blogDir)
    .filter(file => file.endsWith('.md'))
    .map(file => `/blog/${file.replace('.md', '')}`);
}

function generate() {
  const staticPages = ['/', '/auth/login', '/features'];
  const blogPages = getBlogSlugs();
  const allUrls = [...staticPages, ...blogPages];

  const xml = `<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ${allUrls.map(url => `
  <url>
    <loc>${BASE_URL}${url}</loc>
    <changefreq>daily</changefreq>
    <priority>${url === '/' ? '1.0' : '0.8'}</priority>
  </url>`).join('')}
</urlset>`;

  fs.writeFileSync(path.join(__dirname, '../public/sitemap.xml'), xml);
  console.log('✅ Sitemap.xml generated successfully!');
}

generate();
```

12.2 Connecting Business Data for Daily Telemetry

To keep yourself sensitive to cash flow dynamics, you can ask Codex to write a lightweight telemetry script that fetches Stripe's yesterday earnings and new user registrations from Supabase every morning and broadcasts a report card straight to your phone.

Daily Data Report Script (Node.js)

```
// File: scripts/daily-report.js
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
const axios = require('axios');

async function sendReport() {
  // Get the count of new users registered yesterday
  const yesterday = new Date(Date.now() - 24 * 60 * 60 * 1000);
  const userCount = await prisma.user.count({
    where: { createdAt: { gte: yesterday } }
  });

  // Get the count of active subscriptions
  const activeSubs = await prisma.subscription.count({
```

```
    where: { status: 'ACTIVE' }
  });

  const reportText = `📊 [Real-World Product Briefing]\nNew registered users
yesterday: ${userCount}\nTotal active subscriptions: ${activeSubs}\n—— Keep going!`;

  // Send notification to WeChat/Slack webhook
  await axios.post(process.env.MOBILE_WEBHOOK_URL, {
    msgtype: 'text',
    text: { content: reportText }
  });
}

sendReport();
```

By scheduling this on your server's crontab (triggered at 8:00 every day), you create your lowest-cost "business monitoring dashboard."

12.3 The Final Moat: The Only Barrier in the AI-Native Era

When anyone can generate thousands of lines of code in two hours, build a native mobile app, and hook up automated marketing pipelines, **technology itself becomes completely commoditized**. In this new era of "infinite code," the ultimate moat for independent developers and product managers is no longer knowing how to use a specific framework, but rather:

- **Your deep empathy for user pain points (User Empathy).**
- **Your domain expertise accumulated over years in a specific industry (Domain Knowledge).**
- **Your execution capability to leverage AI-native tools (like Codex), launch rapidly, and establish a commercial closed-loop.**

Do not be a mere typist. Go and be the one who defines problems, holds the reins, and resolves real-world pain points.

Thank you for reading the *Codex Blue Book*. Don't forget to bookmark, share, and give a GitHub star! ⭐

Now, configure your `AGENTS.md` in your project root, run your first `codex` command, and go write your own business story.

Ch.13 Frontier Watch: The 2026 Codex Ecosystem Overhaul

> 🎯 ***The Real Problem***: *Rapid tool churn, deprecated commands (Codex desktop merging into ChatGPT code mode), model evolutions (GPT-5.6), and unexpected CLI breaking changes.*
> 💡 ***Tangible Output & Takeaway***: *2026 migration command battle map (winget/brew setups, CLI 0.14x configurations, Plugins ecosystem integration), and compatibility test scripts.*
> ⚡ ***Viral Screenshot Quote***: *"Tools retool every quarter, but mental models remain your moat. Zero fluff—only an actionable battle map of what deprecated and what to adopt."*

The first twelve chapters built a version-independent orchestration methodology. But methodology must land on real tooling. In 2026, the Codex ecosystem went through four structural shifts: **the desktop merger, the model transition, the maturing plugin economy, and security becoming its own product line**. This chapter breaks down each shift with concrete migration commands and configurations.

13.1 The Desktop Merger: Codex App Folds into the ChatGPT Client

In July 2026, the standalone Codex desktop app was retired. All of its capabilities moved into the **ChatGPT desktop client** as a dedicated "Codex Code Mode," available to Free, Plus, and Enterprise users.

Migration steps:

```
# macOS: download ChatGPT.dmg and drag it into Applications
# Windows: install via winget
winget install OpenAI.ChatGPT
```

After signing in, click the **Codex** tab in the left sidebar to enter Code Mode. Two caveats:

1. **Account vs. API Key**: Signing in with a ChatGPT account unlocks the full feature set (cloud tasks, cross-device sync, Computer Use); an API key grants only basic local coding.
2. **Standalone components are unaffected**: the Codex CLI, the VS Code extension, and Codex Cloud continue to evolve independently. The multi-surface matrix from Ch.02 still holds — the "desktop app" cell is simply replaced by the ChatGPT client.

New capabilities at a glance:

- Upgraded built-in browser: search browsing history from the address bar; the Chrome extension can reference open tabs.
- **Multi-repo Review**: multi-folder projects show changed lines across all repositories in one review view — no more tab-hopping between diffs.
- **Sites**: create, deploy, and manage hosted web projects directly in the client, with Annotations for in-place "point and edit" workflows.

13.2 The Model Transition: GPT-5.6 Terra and Luna Take Over

As of August 31, 2026, `GPT-5.4` and `GPT-5.4 mini` are retired from Codex for ChatGPT-authenticated sessions. The official replacements:

Retired model	Successor	Role
`gpt-5.4`	`gpt-5.6-terra` (GPT-5.6 Terra)	Primary deep-reasoning model
`gpt-5.4-mini`	`gpt-5.6-luna` (GPT-5.6 Luna)	Lightweight fast model (Guardian approvals, etc.)

Migration checklist:

```
# 1. Audit your configs for hard-coded legacy models
grep -rn "gpt-5.4" ~/.codex/config.toml .

# 2. Explicitly upgrade the default model in config.toml
```

```
# ~/.codex/config.toml
model = "gpt-5.6-terra"
# Lightweight background duties (e.g. auto-review) can be pinned separately
```

```toml
[profiles.guardian]
model = "gpt-5.6-luna"
```

> ⚠️ ***Note**: API-key-authenticated sessions may continue calling the GPT-5.4 family, but ChatGPT sign-in sessions must switch. Audit every Automation, workspace default, and custom agent.*

13.3 CLI 0.14x: The Breaking Changes You Must Know

The Codex CLI is now in the `0.14x` era (latest stable: `0.147.0` as of August 2026). These changes directly affect every command example earlier in this book:

```bash
npm install -g @openai/codex@latest
```

1. `--full-auto` Is Officially Removed

```bash
# ❌ Old syntax (now errors out)
codex exec --full-auto "fix all lint errors"

# ✅ New syntax: express autonomy via sandbox mode
codex exec --sandbox workspace-write "fix all lint errors"
```

2. The Hooks Engine Graduates to Stable

The interception layer from Ch.05 is now a first-class citizen. Configure `SessionStart` / `Stop` hooks inline in `config.toml`; hooks can observe MCP tool calls, `apply_patch`, and long-running Bash sessions:

```toml
# ~/.codex/config.toml
[hooks]
session_start = "bash scripts/bootstrap-env.sh"
stop = "npm run lint --silent"
```

3. Agent Plugins and Marketplaces

The plugin system is now production-grade, with local, personal, workspace, and remote catalogs — including third-party marketplaces for Amazon Bedrock and Claude Code:

```bash
# Unified plugin management entry point
codex plugin list
codex plugin marketplace add https://plugins.example.com/catalog.json
codex plugin install security-workbench
```

Mention a plugin in chat with `@plugin` to auto-inject its context (MCP / App / Skill).

4. Subagents and Multi-Agent Orchestration

The CLI natively spawns subagents with independent context windows, slicing large efforts into parallel pipelines:

```
> Spawn two subagents: one adds integration tests for src/api,
  another refactors state management in src/hooks in parallel.
  Aggregate the diffs for my review at the end.
```

Combined with `git worktree` isolation, multiple agents work in parallel without polluting each other — the official realization of Ch.04's goal-driven philosophy.

5. Guardian Auto-Approval and `--approve-for-me`

Human confirmation is no longer the only path for high-risk operations. The new `--approve-for-me` flag routes approval requests to a Guardian subagent (powered by GPT-5.6 Luna): in-policy actions pass automatically; violations are blocked with reasons:

```
codex --approve-for-me "upgrade dependencies and make tests pass"
```

This complements Ch.08's mobile approval gateway: Guardian handles the low-risk queue; only genuinely dangerous operations reach your phone.

6. The MCP 2026-07-28 Protocol

Paginated discovery, multi-round requests, and non-blocking server startup are now supported. Legacy `.mcp.json` configs remain compatible, but new MCP servers should target the 2026-07-28 spec.

13.4 The Skills Economy: Compounding Repeated Workflows into Assets

If AGENTS.md is a project's "constitutional law," **Skills are the "standard operating procedures" for specific task types**: PR review, Feishu doc grooming, PPT generation, CI repair, security scanning. Any workflow repeated three times or more deserves to become a Skill.

```
# Standard Skill layout (under ~/.codex/skills/ or inside a plugin package)
skills/
└── pr-review/
    └── SKILL.md      # a procedural manual with YAML frontmatter
```

Skills ship with plugins to marketplaces for uniform team installation. This delivers on Ch.05's promise of cross-project reuse: the protocol constrains behavior; the Skill freezes the procedure.

13.5 Codex Security and Daybreak: Security Becomes a Product Line

In August 2026, OpenAI introduced the two-tier **Daybreak** access system for defenders:

- **Daybreak Blue**: general-purpose models (GPT-5.6 Sol) for vulnerability discovery, secure code review, detection engineering, incident response, malware analysis, and patch validation — start here for most defensive work.
- **Daybreak Red**: purpose-trained models (GPT-5.6 Cyber) for explicitly authorized vulnerability reproduction, exploit validation, penetration testing, and red teaming; requires separate approval and provisioning.

Together with the **Codex Security plugin and CLI**, security workflows are fully productized:

```
# Run a deep scan in the workbench
codex plugin install codex-security
# Or run bulk scans in CI
codex-security scan --deep --report sarif > results.sarif
```

Operational red lines (continuous with Ch.05's sandbox boundaries):

1. Work in an **isolated environment** with an explicitly defined engagement scope.
2. Use least-privilege permission profiles.
3. Configure **Auto-review** for actions that cross the sandbox boundary, so Guardian reviews them first.

13.6 Migration Checklist (TL;DR)

```
# ① Desktop: retire the old Codex App, install the ChatGPT client
winget install OpenAI.ChatGPT        # Windows
# macOS: download ChatGPT.dmg

# ② Upgrade the CLI to 0.147.0+
npm install -g @openai/codex@latest

# ③ Switch models to the GPT-5.6 family (before August 31)
grep -rn "gpt-5.4" ~/.codex/ .       # full audit

# ④ Replace the removed --full-auto flag
codex exec --sandbox workspace-write "<task>"

# ⑤ Enable Guardian auto-approval
codex --approve-for-me "<task>"

# ⑥ Install plugin marketplaces and the security plugin
codex plugin marketplace add <catalog-url>
codex plugin install codex-security
```

13.7 The Constants That Survive Every Version

The tooling lesson is singular: **every version number, CLI flag, and model codename hard-coded into scripts is technical debt**. Centralize them in `config.toml` and `AGENTS.md`, so migration cost collapses to "edit one line."

And the three principles that outlive every release cycle are exactly what this book keeps hammering: drive by goals, not steps; set boundaries before granting trust; divide labor between human and AI — never let the AI lead the human. Tools change blood; your mindset compounds.
